

# Ray casting for incremental voxel colouring

O.W. Batchelor, R. Mukundan, R. Green

University of Canterbury, Dept. Computer Science & Software Engineering.

Email: {owb13, mukund}@cosc.canterbury.ac.nz, richard.green@canterbury.ac.nz

## Abstract

Image based volumetric reconstruction from multiple views is an interesting challenge. Recently several methods of optimisation-based voxel colouring have appeared, which make use of incremental visibility. Culbertson et al. presented a way of determining visibility incrementally by using layered depth images as a data structure (GVC-LDI). We present an alternative algorithm which provides the same outputs. We use ray casting which is simpler and more efficient than using layered depth images.

We make some simple comparisons using rasterized images and look at how it can be applied to optimisation based carving as well as level of detail.

**Keywords:** Scene reconstruction, Voxel colouring, Visibility, Ray-tracing

## 1 Introduction

Voxel coloring [1] and derivatives are methods for reconstructing a scene from a set of calibrated images. A scene is represented by voxels, which are defined to be a unit of volume - in most cases a cube or cuboid. Voxel colouring involves traversing a voxel space from front to back with respect to a group of cameras. A voxel is declared solid. The primary characteristics of voxel colouring are its explicit handling of visibility and its local decision function for solidity, colour consistency.

Space carving [2] introduced the idea of progressively carving voxels to use an intermediate model as a conservative estimate for the true scene visibility information. Space carving is used to support arbitrary view configurations by using a plane sweep method where a mask is used in each image behind the carving plane, to record occlusion.

Generalised Voxel Colouring [3] built on this idea, extending it to support full use of all scene views throughout, with arbitrary view placement. They show two algorithms for computing visibility, the first most simple method (GVC) uses rasterization and depth buffering of surface voxels to update visibility.

The second (GVC-LDI) uses layered depth images of surface voxels as a data structure. As the voxel model is updated the layered depth images are also updated. The key point is that the visibility is updated incrementally and only as required. The basic GVC algorithm suffers from needless re-calculation as the algorithm becomes close to converging. We look at an extension of Generalised Voxel Colouring [3] as an alternative to using layered depth images we consider a data structure of

image rays, which is efficiently updated as a voxel model is carved.

We make use of voxel raytracing to update voxel visibility. Voxel raytracing is a well known technique and there exist algorithms for traversing various structures, a regular grid [4] as we make use of here or an octree [5]. Voxel raytracing is typically used to traverse spacial subdivision structures in raytracing or collision detection, or for volumetric ray tracing.

Voxel colouring has distinct advantages over silhouette based reconstruction, it can take into account "interior" silhouettes, as well as texture information. Optimisation-based colour consistency is an attempt to make better use of texture information, as threshold parameters are highly dependant on the scene texture.

Optimisation-based colour consistency involves evaluating if a carving operation will improve some global function (e.g.. reduce reprojection error). If a carving improves the global function then the voxel is carved. This can be reduced to a local decision once again by assuming the pixels in projections of each voxel are independent. Recently two methods based on this idea have arisen, one using a framework for adding and removing voxels to attempt to improve reprojection error [6], another using a pixel centered statistical model with carving [7]. Both methods use GVC-LDI to evaluate visibility.

## 2 Voxel ray casting

The basic idea is to use an image of rays as a data structure. The rays passing through an image plane can be sampled at discrete intervals - one

through the centre of each pixel. An image of rays can be created from the projection matrix of pinhole modelled camera.

At each pixel/ray we store the information required to traverse the voxel grid incrementally. This is done in the form of the information required for an iteration of voxel raytracing [4], and includes direction and the distance to the next intersection on each axis (as well as the current voxel). Each ray is initialized to the first hit solid voxel.

First we give an overview of common definitions which we have used the names given in [3] for consistency (e.g.  $\text{Vis}(V)$ , CVSVL). The volume is represented by a 3D binary image. The surface is represented by a hashed set of surface voxels. As the volume is carved, the surface is updated. When a surface voxel is removed, it uncovers interior voxels which are then added to the set of surface voxels.

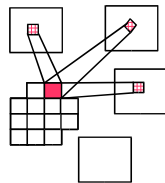


Figure 1:  $\text{Vis}(V)$  - Visible projections of Voxel  $V$

Generalised voxel colouring algorithms [3] work by maintaining a Surface Voxel List (SVL) which is used as a conservative estimate of the true scene for visibility calculation. Only the visible pixels ( $\text{Vis}(V)$ ) in each image are used at each step. Voxels are evaluated for colour consistency as a function of visible pixels  $\text{Consist}(\text{Vis}(V))$ . If a voxel is not colour consistent it is carved, and the SVL and visibility structures are updated accordingly.

$\text{Vis}(V)$  is defined as the set of visible, projected pixels for one voxel, onto each image - shown in figure 1.  $\text{Vis}(V)$  represents the visibility information for one voxel.

$\text{Vis}(V)$  can be calculated by scanning an Item buffer. An item buffer is an image which stores the ID of the closest voxel at each pixel. The potential pixels to be examined can be identified by rasterizing the voxel to each image.

GVC creates an Item buffer by rasterization and depth buffering, GVC-LDI uses the head of each voxel list as an Item buffer. We use an image of rays as an Item buffer, with each pixel we store the nearest visible voxel to each ray.

In practice, due to small differences between the set of pixels projected by the rasterization algorithm

and ray casting algorithm we instead examine an area within the bounding box of a voxels projected vertices.

Carving proceeds in a serial process as per GVC-LDI. The Changed Voxel Surface Voxel List (CVSVL) is a set of voxels which require evaluation for colour consistency, because their visibility has changed. It is initialised to the surface list. Colour consistency is evaluated as before from  $\text{Vis}(V)$  as  $\text{Consist}(\text{Vis}(V))$ .

```

Initialise Volume, Surface,
                CVSVL, Visibility

while(CVSVL is not empty) {
    Remove voxel V from CVSVL
    Calculate  $\text{Vis}(V)$ 
    if not  $\text{Consist}(\text{Vis}(V))$  then {
        Carve V
        Update Surface
        Update Visibility
        Add changed vis. voxels to CVSVL
    }
}

```

Figure 3: Incremental voxel colouring overview

Here we show the common factors between incremental voxel colouring algorithms 3, the differences when using ray casting are in the steps for updating visibility.

When a voxel is carved, each ray pixel in  $\text{Vis}(V)$  is stepped until the ray intersects another solid voxel (which will either be a surface voxel, or just about to become a new surface voxel). Voxels with changed visibility (those which are uncovered by rays) are added to the CVSVL.

```

For each pixel P in  $\text{Vis}(V)$  {
    do {
        step Ray(P)
    } until solid(Voxel at Ray(P))

    add Voxel at Ray(P)
    (if any) to CVSVL
}

```

Figure 4: Visibility update for ray casting

The algorithm given in 5 is slightly more brief than given in [3], this is because we have left out the process of updating the surface which they have incorporated in the same piece of pseudo-code, however it is equivalent - and even contains an optimisation that only newly added surface voxels need updating rather than all neighbours.

The main advantages of this approach to updating visibility lie in the simplicity, it does

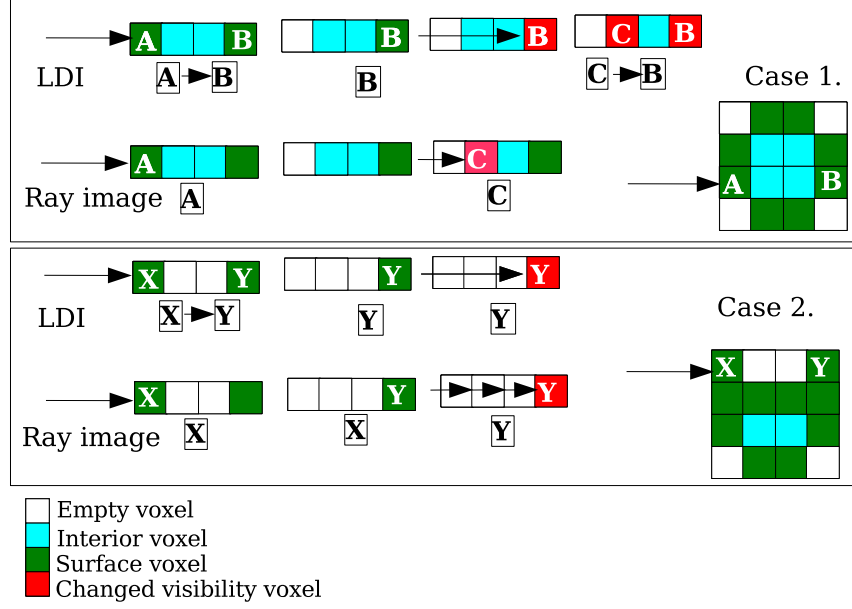


Figure 2: Updating visibility for LDI and Ray image

```

For each pixel P in projection
  of V onto all images {
    Remove Voxel V from LDI(P)
    add next Voxel on LDI(P)
      (if any) to CVSVL
  }

For each new neighbour N {
  For each pixel P in projection
    of N onto all images {
      Add N to LDI(P)

      if N is the head
        of LDI(P) add N to CVSVL
    }
  }

```

Figure 5: Visibility update for GVC-LDI

not require computing any projections beyond the  $\text{Vis}(V)$  which is required in any case to evaluate consistency -  $\text{Consist}(\text{Vis}(V))$ . It uses a static data structure which does not require overhead of dynamic allocation or depend on any property of the scene.

While GVC-LDI is roughly  $O(\text{average list size} \times \text{voxels})$ , using ray casting is roughly  $O(\text{average steps} \times \text{voxels})$ , given constant voxel and image resolution - however issues of practicality seem to out weigh theoretical complexity.

Figure 2 shows the pixel-level details as to the differences between the two. It shows two cases, case 1 where a voxel is carved from a solid block, case 2 where a voxel is carved just touching the edge of the surface (from the perspective of the view, one pixel of which is shown by the arrows). The current state of the LDI/closest voxel is shown at each step.

In the first case voxel A is carved, and removed from the LDI, voxel B is next in the list, so it is added to the CVSVL, voxel C is then added as a new surface voxel, thus it is also added to the CVSVL. In the case of ray casting, the ray is just stepped once until it reaches another solid voxel, voxel C - which is added to the CVSVL.

In the second case voxel X is carved, and removed from the LDI, voxel Y is next in the list, so it is added to the CVSVL. In the case of ray casting, the ray must be stepped three times until it reaches solid voxel Y, which is added to the CVSVL.

It seems that preferably to use the ray casting method in case 1, the LDI method in case 2 - however the LDI method does much worse in practice because of the extra rasterization involved, and case 1 is more common for typical scenes.

### 3 Comparisons

We have implemented GVC-LDI, GVC and a carver using ray casting to compare using a simple experiment. We captured 14 images at 800x600 surrounding a rasterized scene of a textured teapot, a sphere and a cube sitting on a checkered

Table 1: Runtime statistics for an earlier trial with 12 images

Method	Convergence time	Memory use	Number carved	Number evaluated
Ray casting	34.68s	230Mb	1218646	1763693
OpenGL-GVC	57.77s	60Mb	1214362	6724928
GVC	158.66s	75Mb	1220041	6603412
GVC-LDI	401.54s	320Mb	1217602	2532613

grid. Reconstruction was performed for a  $120^3$  volume enclosing the scene. We used a Pentium 4 CPU 2.80GHz, with 1Gb of ram.

A simple consistency function is used with all methods, based on the difference of the mean colour from a view to the mean voxel colour, weighted by number of pixels.

Most parameters are controlled, however some are inherent in the differences between methods - such as the carving order, and the exact projections which varies between OpenGL and software rasterization. Despite minor differences, all methods produce very similar output and the trends are shown well.

### 3.1 Implementation details

Of the two versions of GVC one is implemented as closely as possible to [3], as is the implementation of GVC-LDI. A second implementation of GVC is simplified to use OpenGL for hardware rasterization and updates all visibility information and colour statistics simultaneously by scanning pixels of the item buffer and updating a giant table of voxel x view statistics.

All implementations share some common base code which is very slightly optimised for rasterizing cubes (e.g.. backface culling, interior face culling) and common data structures for surfaces, volumes. We have tried to ensure there is no bias towards implementation details by profiling runtimes and eliminating bottlenecks.

### 3.2 Results

Figure 6 shows the convergence patterns of each method, of note are the two different forms - the GVC algorithms rapidly decrease rate as convergence nears, where the two incremental methods GVC-LDI and ray casting are both nearly completely linear (despite the GVC-LDI curve being trimmed for space). The ray carving method shows a factor of almost ten fold reduction in convergence time over GVC-LDI.

While ray tracing is seen as incredibly slow compared to depth buffering - it is important to note that GVC-LDI uses a complicated data structure

maintaining all surface intersections in a linked list per pixel. Maintaining this structure requires considerable effort using multiple rasterizations per carving and linear searches to update linked lists at each pixel. The raycasting approach uses a static sized data structure and requires just one rasterization per carving.

The ray tracing involved in stepping each individual ray requires miniscule time compared to the rest of the process. As an illustration - an earlier effort was to pre-calculate all voxel intersections per ray. This proved slower to update than ray casting when all that was required was to pull numbers out of a container.

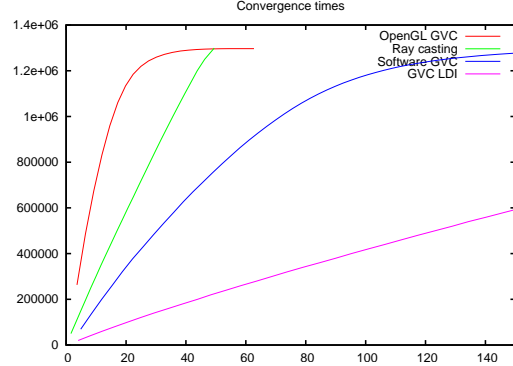


Figure 6: Time vs voxels carved for several methods

Table 1 shows the runtime statistics for another earlier experiment where memory use was monitored. Memory requirements of GVC-LDI and ray casting are similar, and both large in comparison to variants of GVC. Also of note is the number of evaluations, which is roughly three times as high for GVC methods and lower for GVC-LDI, and slightly lower for ray casting. While the colour consistency function used here contributed only a tiny portion of the time (less than 10% for ray casting), ray casting may be beneficial when using a more computationally intensive function.

## 4 Application

### 4.1 Optimisation-based voxel colouring

The motivation for this work was to provide a base for optimisation-based voxel colouring. We have



(a)



(b)

Figure 7: (a) Input image of test scene (b) Image of reconstructed voxel model

attempted to verify the method of two prior works. Firstly a reduced version of optimising reprojection error [6] using carving only, secondly [7] claims to produce very good results.

An optimization-based algorithm will require information on voxels which will change visibility, such as their colour - depending on the function used. This can be updated incrementally by adding newly visible pixels to a mapping of surface voxels to an accumulator. We have used this for re-ordering the carving process by treating the CVSFL as a priority queue ordered by the number of visible pixels.

## 4.2 Level of detail

Due to the rapid convergence rate of incremental methods, this makes them ideal for use with level of detail or pre-carving with silhouettes. We have looked at it's use with an idea presented in [8] which augments (expands) the voxel space before carving at increasingly higher resolutions. These greatly reduce the work in carving when far from a global solution.

Initialise voxel space at low resolution

```
while(Resolution less than desired) {
    Carve voxel space
    Augment voxel space
    Increase resolution of voxel space
}
```

Figure 8: Simple level of detail algorithm

Using a level of detail algorithm such as this improves time taken dramatically. A reconstruction involving 50 images of 800x600, at  $240^3$  using four carving steps beginning from  $30^3$  may take just

under 5 minutes, which may otherwise take over an hour. This comes potentially at the loss of some accuracy as lower resolution reconstructions may be over-eager, destroying parts of the model.

## 4.3 Conclusion

We have presented a method using ray casting as an alternative means to calculating visibility for incremental voxel colouring. It has several advantages over the algorithm presented by Culbertson et al. in [3] it is simpler and more efficient.

In simple experiments on synthetic rasterized scenes a voxel colouring algorithm based on ray casting converges at a time of a factor of 10 faster than an implementation of GVC-LDI. Reasons for this include using a static data structure (as opposed to linked lists) and requiring fewer voxel rasterizations per carving.

We have applied this method to optimisation-based voxel colouring algorithms, and level of detail. Currently we are working on a simpler technique which stores rays and voxels in a more direct way bypassing image projections.

In future we would like to focus on improving reconstruction quality and evaluation with real scenes. Some ideas are modelling smoothness for regularization and view dependant voxel colour in the framework of optimization-based voxel colouring.

## References

- [1] S. M. Seitz and C. R. Dyer, "Photorealistic scene reconstruction by voxel coloring," in *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition*

- (*CVPR '97*), p. 1067, IEEE Computer Society, 1997.
- [2] K. N. Kutulakos and S. M. Seitz, “What do n photographs tell us about 3d shape?,” January 1998. Computer Science Dept. U. Rochester.
  - [3] W. B. Culbertson and T. Malzbender, “Generalized voxel coloring,” in *Proceedings of the ICCV Workshop*, vol. Vision Algorithms Theory and Practice, September 1999.
  - [4] J. Amanatides and A. Woo, “A fast voxel traversal algorithm for ray tracing,” in *Eurographics '87*, pp. 3–10, Amsterdam, North-Holland: Elsevier Science Publishers, 1987.
  - [5] J. Revelles, C. Ureña, and M. Lastra, “An efficient parametric algorithm for octree traversal.”
  - [6] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafer, “Improved voxel coloring via volumetric optimization,” 2000.
  - [7] H.-W. Kim and I. S. Kweon, “Optimal photo hull recovery for the image-based modeling,” in *The 6th Asian Conference on Computer Vision (ACCV)*, Jeju, Korea, January 2004.
  - [8] A. C. Prock and C. R. Dyer, “Towards real-time voxel coloring,” in *Proc. Image Understanding Workshop*, pp. 315–321, 1998.